Show Design and Control System for Live Theater

by

Michael R. Miller

S.B. C.S., M.I.T., 2009

S.B. Music, M.I.T., 2009

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 2010

[June 2010]

Author_____
Department of Electrical Engineering and Computer Science
May 17, 2010

Certified by_____
Tod Machover, Professor of Music and Media
Thesis Supervisor

Accepted by_____
Dr. Christopher J. Terman
Chairman, Department Committee on Graduate Theses

Show Design and Control System for Live Theater
by
Michael R. Miller

Submitted to the
Department of Electrical Engineering and Computer Science

May 17, 2010

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Tod Machover's upcoming opera *Death and the Powers* calls for an unprecedented integration of technology and theater. These uses of technology are at the heart of the show and include a Greek chorus of nine "Operabots"; a novel surround sound and audio processing architecture; a massive string instrument called "The Chandelier"; three 14' tall robotic stage fixtures with displays called "The Walls"; and more. Each component has its own unique challenges, but all share the need for a comprehensive show design and control system. A show design and control system was built for this specific purpose and will be used for *Death and the Powers.* The software affords real-time control over the robotic elements in the opera alongside predetermined, choreographed routines.

Thesis Supervisor: Tod Machover
Title: Professor of Music and Media, MIT Media Lab

Table of Contents

Table of Figures

1. Background

1.1 Problem Definition

Tod Machover's upcoming opera *Death and the Powers* calls for an unprecedented integration of technology and theater. The technological assets include a Greek chorus of nine "Operabots"; a novel surround sound and audio processing architecture; three 14'-tall robotic stage fixtures with display surfaces called "The Walls"; and more. Each technology presents its own unique challenges. Each also demands integration with the other technologies and with the more traditional features of an opera: writers, composers, actors, musicians, choreographers, designers, and directors.

The theatrical process is composed of roughly two aspects: design and production. The design aspect of an opera includes many different subprocesses such as the writing of the libretto; music composition; choreography; lighting design; visual design; and more. The production aspect focuses on the physical enactment of the design, both in rehearsals and in performances. When new technologies are used in theater, they must be part of the design and production of the show, and their creators must find a way to make them all work together.

Since the focus of this thesis is technical in nature, the following overview of the opera itself and the countless other detailed design and production decisions are described in general terms. This brief account does not adequately express the creativity and effort of the many talented individuals that have gone into the show. Additionally, because this thesis was written before the premiere of the show, a full account of the design and development of *Death and the Powers* is not yet available. Interested readers should visit the *Death and the Powers* website[1]. Detailed information on the various technology elements in the opera—as well as a more in-

[1] http://opera.media.mit.edu/projects/deathandthepowers/

depth history and description of the opera itself—is available in Peter Torpey's master's thesis.

1.1.1 Death and the Powers

Technology is at the heart of *Death and the Powers.* The story, conceived by Randy Weiner and former Poet Laureate Robert Pinsky, revolves around the family of Simon Powers, a wealthy and powerful entrepreneur and inventor. Nearing the end of his life and looking for a way to prolong and expand his existence, Powers decides to transfer his "being" into the technology he created and fostered: The System. After he does so, his family struggles emotionally with his transformation as they look for signs of life in spite of his physical absence. The opera explores not just how technology affects our lives but what it means to be alive.

The story of Simon Powers and his family is bookended by a prologue and epilogue set sometime in the future. In this future, a troupe of robots are recounting and enacting the story, trying to understand what it is to die and to be alive. These robots are expressive and sentient, and possess a child-like naïveté and curiosity towards what it means to be human.

The opera appropriately includes many novel technologies, given the story's roots in technology. These technologies include the Walls; the Disembodied Performance System; and the Operabots.

The Walls are three 14'-tall robotic periaktoi[2] (Figure 1). These massive triangular set pieces feature addressable LED display surfaces and static faces mimicking bookshelves. Additionally, each face can be backlit. The Walls serve a dual purpose

[2] A periaktos (periaktoi, plural) is a triangular prism-shaped set piece dating back to theater productions in ancient Greece. Different faces would have different scenes, so rotating the periaktos might constitute a scene change.

as set pieces and as components in The System. Often, the display surfaces will be used to evoke Powers' presence and personality.

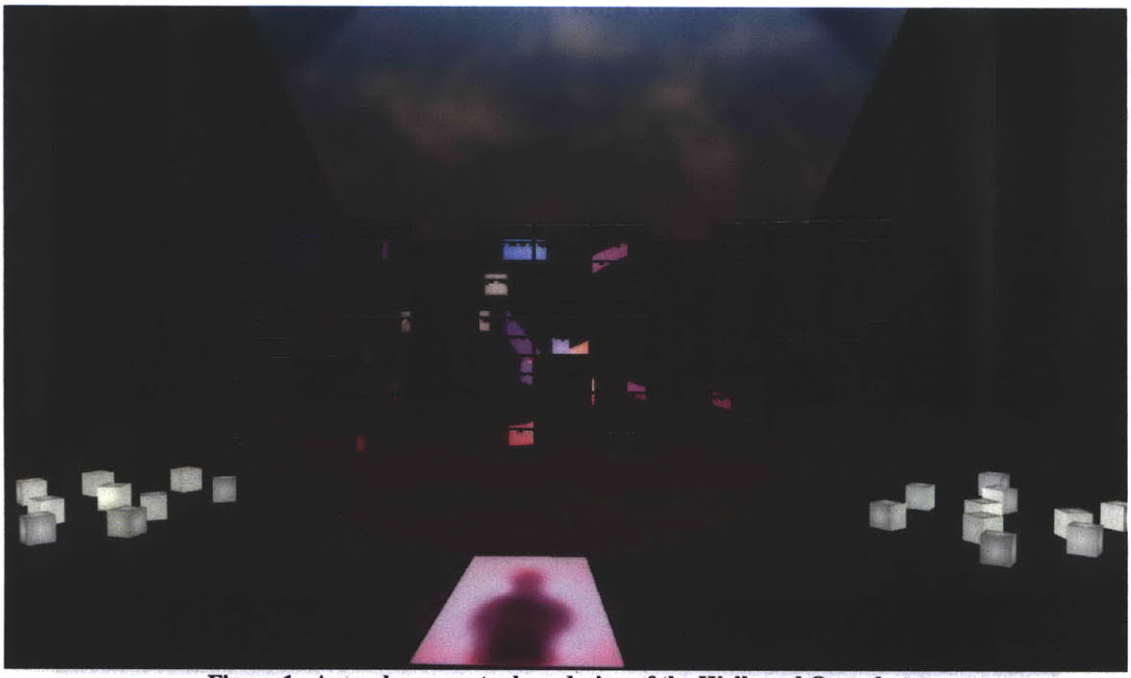

Figure 1. An early conceptual rendering of the Walls and Operabots
(by Peter Torpey)

Peter Torpey's Disembodied Performance System allows for the projection of Powers' persona onto the various elements of The System. It employs sensors that measure the actor's breathing and movement. The Disembodied Performance System interprets and conveys emotional intent using this sensor data. One goal of the Disembodied Performance System is to provide a meaningful visual embodiment of Powers in the Walls.

The Operabots are another component of The System. They act and dance in the show alongside the singers. The Operabots feature prominently in the show's prologue and epilogue and perform an elaborate dance routine in the middle of the show. At other times, they are still present but not in the foreground.

The Operabots recall the Walls with their triangular shape (Figure 2). They can turn in place and move in any direction. They stand between four-and-one-half and seven-and-one-half feet tall, depending on their elevation height. Designed by production designer Alex McDowell and realized by Bob Hsiung, Karen Hart, and Jason Ku, they are built from a variety plastics and metals. Each Operabot has sixteen degrees of control including three omni-drive wheels; eleven addressable channels of lighting; tilt on the head; and elevation on the midsection. The omni-drive wheels allow for fluid motion in any direction. Bright white LEDs are used as indirect and direct lighting elements.

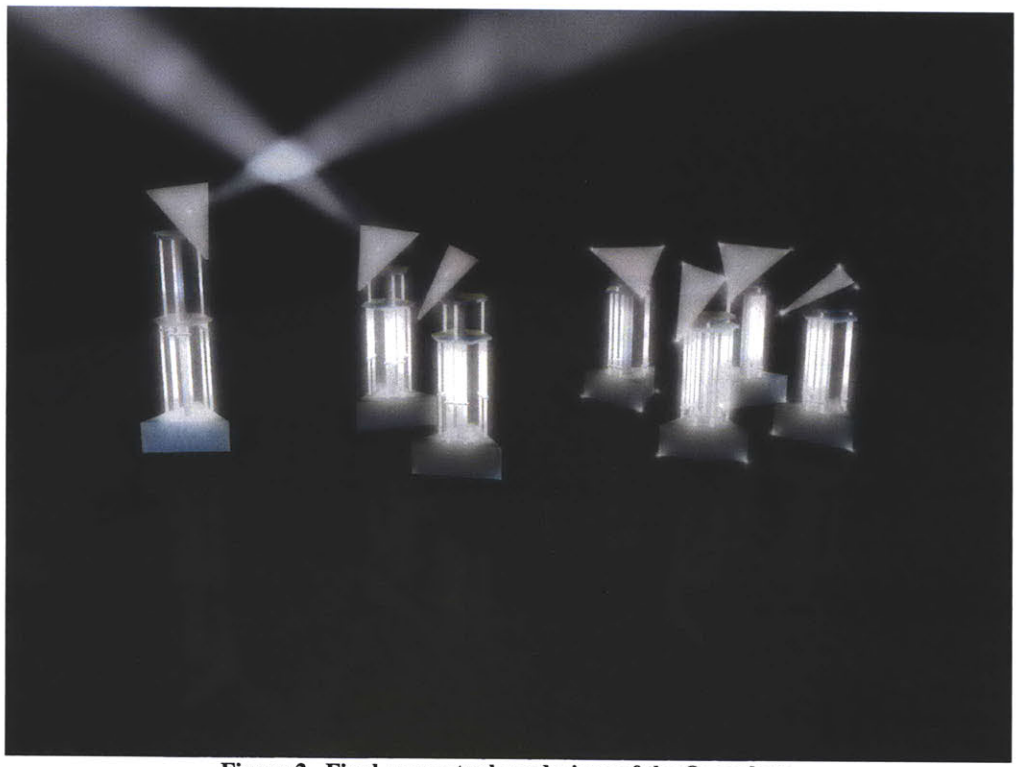

Figure 2. Final conceptual renderings of the Operabots
(by Peter Torpey)

The eleven lighting channels are situated around the robot. There are two sets of three lights at the corners of the head and base respectively, each set controllable as a group. Two channels control strips of LEDs in the head and base that provide indirect lighting through the translucent plastic exterior. The nine acrylic rods in

the middle are broken into three lighting groups. The three leftmost rods on each side of the Operabot constitute one group. The center and rightmost rods are similarly grouped. Within each of these groups, the lights at the top and bottom of the acrylic rods are separately controllable. The rods thus account for six channels of lighting. Lastly, there is an extra bright, white LED spotlight recessed in the center of the head, aimed upward.

1.1.2 Development Process

Death and the Powers has been in development for over ten years. During that time, visions of every piece of technology in the show have evolved to meet limitations in budget, time, and feasibility. For example, initial designs for the Walls called for a physical realization of a giant bookcase over twenty feet tall. Each of the books—more than a hundred—could be individually actuated with more than three degrees of freedom and housed multiple RGB LED "pixels" to provide a display surface. This turned out to be physically impractical and extremely costly, especially when other methods could achieve similar effects more easily. As well, when the venue for the show's premiere changed, the Walls were made smaller to accommodate the new stage size.

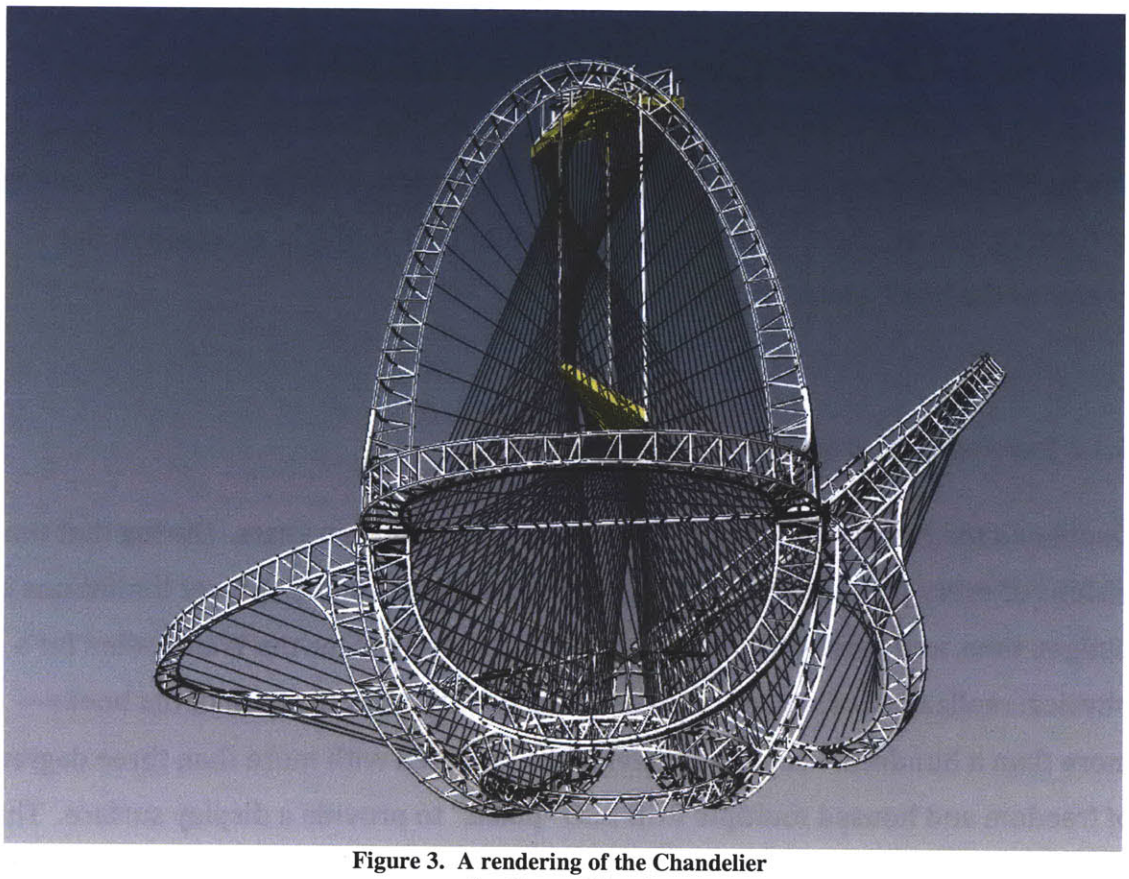

Figure 3. A rendering of the Chandelier
(by Steve Pliam)

The show also features a giant hanging set fixture called the Chandelier (Figure 3). Initial plans called for a massive string instrument with seven-foot strings. It was to be played both remotely by an instrumentalist (perhaps on a keyboard) and locally by another musician, perhaps by plucking or damping strings. Besides being a stunning lighting piece during the show, it also plays an integral part in the drama during a sexually charged scene between the disembodied Simon Powers and his wife Evvy. The Chandelier—an extension of Powers' "being" in The System—descends and encompasses Evvy. Powers and Evvy perform a vocal duet and connect physically through the strings of the Chandelier.

Years of research by various graduate students under Tod Machover's direction prototyped different designs. Unfortunately due to time constraints, the instrument itself will not be ready in time for the opera. As such, the instrument component of

the Chandelier has since been removed from the design destined for the show. It remains an instrument in the libretto, however, so the orchestra will emulate the sound during the actual performance.

Early designs for the Operabots looked and acted very differently (Figure 1). A collaboration with the Personal Robots Group at the Media Lab produced a small, foot-wide, cube-shaped prototype with omni-drive wheels and precise positioning using a Vicon tracking system [1]. Another prototype of the Operabots closely resembled the current design but relied on two drive wheels and casters instead of omni-drive.

As the show nears production, the creative ideas and engineering have coalesced into a final design. The Operabots were the first to reach near-final form, with the prototype from 2008 almost identical visually to the 2010 production design. The switch to omni-drive wheels is the only significant functional difference. The Walls' individual motorized books were replaced by a display surface of tubes of RGB LED pixels, masked at varying heights to create the visual illusion of books. The Chandelier, no longer a functional instrument, has been reconceived and will be either a non-musical set piece or a music controller that triggers sampled or synthesized sounds based on interaction with the strings.

1.1.3 Show Design and Control

The disparate robotic elements in the show share a common requirement: show design and control software to craft and to coordinate varying modes of operation. The choreography for the Operabots in the prologue differs substantially from the routine in the epilogue, for instance. From scene to scene, the Walls need to move into different arrangements on stage. As choreographers and lighting designers create dance routines and light shows for the Operabots, and the director blocks out the Walls' movements, these directions need to be translated into a form

communicable to the robots. These types of interactions necessitate a user-friendly design toolkit that allows rapid prototyping and iteration of designs.

The show control software is responsible for sending the stage directions to the robots in real-time. It acts as a device server, relaying commands to the various connected components and assuring their safe operation during the performance. Other show control systems in typical use in theaters include lighting boards and audio mixing boards. These two show controllers ensure safe, reliable, and repeatable manipulation of lights and sound during the performance. The robot show control should work analogously, providing robust guarantees for safety, reliability, and repeatability.

1.1.4 Requirements

Since many of these technologies have never been used in a theatrical show before, the requirements are quite broad. Additionally, the incremental changes and improvements in the design of the show have resulted in general solutions, ready to adapt to unforeseeable shifts in vision.

The robots are intended to be as autonomous as possible. Physically bounded elements like lights (which fortunately cannot wander off-stage) can easily be autonomous. But achieving autonomous, unconstrained, and safe movement with seven-foot tall robots is a difficult feat. Moreover, artistic directions call for intricate choreographed dances between up to nine Operabots at a time, weaving among each other and the performers across the stage. This movement is impossible purely with dead reckoning. Instead, such intricate motions call for a precise absolute positioning system. The early box-shaped Operabot prototypes accomplished this absolute positioning through a Vicon infrared tracking system. The latest plans employ an Ubisense ultrawideband (UWB) real-time tracking

system [2]. Robots, set pieces, and performers will be outfitted with tags to provide precise location data to various subsystems, including the show control system.

Some scenes require more complex expressivity than a semi-static routine can provide. The system must be capable of supporting multiple operators who can manually control individual robots using various input methods such as joysticks, gamepads, or touch screens. Current plans call for four operators. These operators also act as a safety net, monitoring the behavior of the robots and preventing calamitous situations through manual intervention.

Due to the creative and ever-changing nature of art and theater, the design software must be able to rework entire scenes rapidly to meet adjustments in artistic direction. Most of the choreography will be completed before rehearsals for the performance even begin, but some routines may change on-the-spot during the rehearsals or even during the performance if, for example, a performer misses a cue.

As a creative tool, the design software needs not only to allow the user to realize his or her imagination, but also to encourage it. During the design process, the software must be capable of presenting the choreography in an intelligible fashion. A real-time visualization of the robots is the most effective way of realizing the user's input. This visualization would allow the user to preview the outcome without using physical robots. Thus, the artist can quickly draft new scenes and receive reliable feedback behind the confines of a desk, without access to a large stage and many robots.

The show control system must be similarly flexible. During rehearsals, scenes will often need to be run over and over. While it is natural for actors to stop and go back to their positions, this is a little bit more difficult for robots. The robots need to return to their starting places efficiently and safely without running into anything. Stop-and-go playback of scenes must be straightforward.

Safety requirements are stricter in a live theater production than in a general usage environment. First and foremost, the system must be robust to all imaginable and unimaginable modes of failure. The software simply cannot break. At best, if the show control fails, a troupe of two-hundred-pound Operabots will grind to a halt and be dragged offstage by stagehands in the middle of a performance. At worst, the robots will continue to drive off out of control and cause damage to performers, audience members, and property. Even the "best" situation is unacceptable.

There are two primary components to safe operation: detecting unsafe conditions, and handling unsafe conditions. Detecting unsafe operation can be difficult. Active communication between all levels of components is a necessity for safe operation. Communication-based timeouts are an effective way of detecting errors. If a system does not respond to a communication within a certain time interval, then it is presumed to have failed. For instance, if an Operabot does not hear from the show control within a few seconds, something may be wrong with the show control. Or if the show control does not hear acknowledgements from a given Operabot, then something is presumed amiss with that particular Operabot. Such a methodology will not determine the cause of failure, but it will reveal which level in the hierarchy failed, be it hardware, software, or the network. Without knowing the underlying cause, any unsafe situation should be treated as a potential catastrophic failure.

There are many possible modes of unsafe operation, including hardware failures, software failures, and physical failures. Hardware failures include loss of control, overheating, or a complete hardware malfunction. Detecting a hardware failure may be as simple as recognizing a loss of communication between the hardware and controlling software. If the hardware stops acknowledging commands from its controller software, it has potentially failed. Software may fail through a bug; crashing; malfunctioning computer hardware; or network-related issues. It is up to

the software or hardware that connects to the failed component to detect the malfunction.

Alternatively (or at the same time), safety issues may be physical in nature. Since the Operabots are tall and rather awkwardly balanced, a sudden deceleration might cause an Operabot to tip over under the right circumstances. More importantly, an Operabot might run into another object. In their current form, the Operabots have no sensors on them. As such, there needs to be some mechanism by which a robot can detect an object in its path. In this regard, the external Ubisense positioning system is a key piece of equipment for safety. Since performers, Operabots, Walls and other stage elements are outfitted with tracking tags, the Operabots can identify potential collisions. Under no circumstances is it acceptable for an Operabot to come into contact with another object.

Tiered fail-safes—from the low-level hardware that controls the drive motors to the high-level show control software—can be used to handle failures. In the event of hardware failure such as an unresponsive motor, the controlling software should react by first attempting a safe stop. Then, it should report its error up the chain of command until it reaches the show control. At this point, it is the operator's responsibility to decide what happens next.

Less severe errors may not need to propagate up the entire chain of command. For instance, suppose an Operabot is aware that it will hit an object if it continues on its current trajectory. It should stop and wait for operator interaction. However, if that object moves from the robot's path, the robot should recover gracefully and continue as instructed previously.

It is impossible to detect all forms of failure and unsafe operation. Detecting physical breakage on the robot is problematic, for instance. In such cases, it is up to the operator to stop operation manually. Manual kill switches at every level of

hardware and software are necessary to deal with problems not directly detectable by the automated fail-safes.

1.1.5 Summary of the Problem

The problem, then, is to create software to design content and run the show. This content includes choreographed routines not just for the Operabots but also for the Walls and potentially other robotic and lighting elements. This software needs to be accessible and flexible, as well as encourage creativity and a rapid design process. The software must control these same elements during the rehearsals and performances, following both the designed routines and direct operator interaction. Finally, this software should ensure safe operation while conveying the design as accurately and reliably as possible.

1.2 Prior Work

Most research in robotics tends to focus on autonomous uses with a large emphasis on artificial intelligence, computer vision, and path finding. This particular usage in theater requires a more controlled solution. Rather than being purely autonomous or human-operated, the robots will be controlled primarily via predefined choreography (automation). Few existing tools target robot automation outside of a manufacturing context. Software like The Player Project [3] and Microsoft Robotics Developer Studio[4] provide a feature set including a visual simulation mode and network control of devices. However, neither is intended for automation or the rigors of live theater environments.

The most similar piece of existing software is Stage Technologies' eChameleon[5]. eChameleon is a proprietary software package used by Cirque du Soleil and other massive theater productions for large scale automation ranging from moving lights

to motorized stages. Like the proposed system, eChameleon provides a 3D simulation for planning the show. Additionally, it provides direct integration with Autodesk's 3ds Max [6] modeling software for complex animations. Similarly, Medialon's Manager [7] is the industry-standard show control tool. While it would provide precise control over the bounded axes of control like the lights on the Operabots, it would have struggled with unbounded parameters like the position of the robot.

The user interface for the show design software draws inspiration from entertainment-software idioms as used in Apple Final Cut Pro [8], MOTU Digital Performer [9], and similar video- or audio-editing applications. This software is intended as an extension of such concepts applied to show design and control for robots.

2. Methods

This document details a software solution which combines the show design and control aspects into one application. This show design and control system is responsible for controlling the many robotic elements in *Death and the Powers*, such as the Operabots and the Walls. Not only does it provide live control over the robots and the show's progress, but it also provides tools to design the automated movement of these robots throughout the show.

The software system has two primary functions—it is both a design toolkit for the show and the show control system itself. While both subsystems are highly connected, they have differing goals and requirements.

2.2 General

In the show design and control system, the show is broken down into a number of sequential scenes called cues. These do not necessarily line up with theatrical scenes in the opera itself. Rather, they represent moments of system-wide realignment when the Operabots and Walls enter a new mode of operation. For instance, the Operabots may be stationary with blinking lights in a cue but begin to move as soon as the next cue is triggered.

Cues manage a number of controllable devices (Figure 4). Each device has sub-devices and axes of control. Cues then route a signal to a given axis of control. An Operabot, for instance, is a device, perhaps comprised of a drive sub-device and a lighting sub-device. The lighting sub-device consists of all eleven addressable lighting elements on the robot. The drive sub-device may have axes for X and Y speed, or more generally, absolute X and Y position. This hierarchy naturally abstracts away the specifics of the underlying hardware in a convenient manner. As

such, the Walls, the Operabots, and any other elements are represented in a consistent, similar fashion.

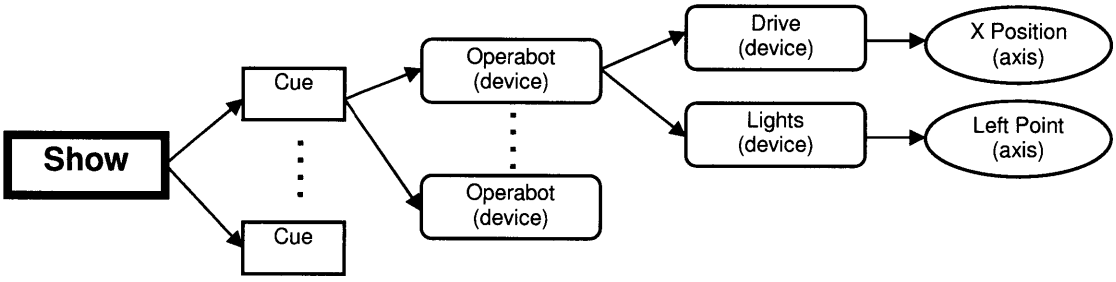

Figure 4. A simplified example hierarchy for a show including cues, devices, subdevices, and axes

Automating axes is the predominant mode of control, but there are two alternatives to predefined animations: live control and programmatic control. Live control allows an operator to manipulate a device with a joystick or a gamepad. Live control simply maps an axis on the input device, such as the X axis of the joystick, to an axis on the output device. Programmatic control allows the user to map the results of programmatic routines to the various output axes. For instance, one could have all the Operabots turn to follow the position of another Operabot on stage.

2.2 Design Subsystem

The design subsystem allows rapid prototyping of choreography for the robots. The tools must be easy to use and provide a clear representation of what the final output should be. To this end, the design toolkit is accompanied by a 3D visualization of the robots and the stage called the Simulator. The Simulator animates the devices in the show according to the axes specified in the design. Real-time playback, like in video- or audio-editing software, allows the user to preview the results without the risk or hassle entailed by using physical robots.

Users can easily manipulate the automation signal for each axis. As well, the same signal may be shared in multiple places in the same or different cues. Tweaking that

one automation signal means the rest will update accordingly—unless that is undesirable, in which case one can duplicate the data. Signals also have "end behaviors" which extend the signal infinitely after the last user-defined value. End behaviors include stopping (setting the signal to some off value); repeating (starting again from the beginning of the signal); mirroring (alternating repeating from the end to the start and normally); and more.

Each axis is represented by a track. Tracks are composed of a linear sequence of clips rather than the raw signals. Clips are a finite, framed view of the underlying infinite automation signal.

Clips are aligned against a cue-wide, shared timeline. Clips have a start time and duration along that timeline. The value of an axis at a given time is the value of the underlying signal at that time within the clip. At any point along the timeline where no clip is active, the axis is considered off. Clips allow the user to isolate a subsection of a signal: one need not start from the beginning of a signal or end at exactly the end. Clips are a beneficial abstraction for the underlying signal: they facilitate the high-level reordering and reuse of signals throughout the show.

The other main element of the design tool is the Simulator. The Simulator provides a 3D simulation of all the axes and devices in a cue. Like in audio- or video-editing software, the design tool supports playback. When in simulation mode, that playback produces a virtual preview of the show. This allows users to quickly create content and get a sense of how the results might look even before the physical staging and rehearsals for the show. The simulation window also acts as a controller, allowing one to plot movement for the various devices.

The Simulator is not simply a visualization. Each device in the simulation uses the actual software that runs on the physical device, but with virtual hardware. The software is unaware it is sending commands to a 3D rendering instead of a motor.

This reuse of the on-board software also makes the Simulator a very useful tool for testing the control software on the Operabots and other devices.

The ability to jump around in the timeline that worked so fluidly in simulation does not transition as smoothly to the physical world. However, it is a necessary feature for rehearsals when scenes need to be run out of order and many times in a row to get things right. As such, the ability to move to a specified point in the timeline is a must! To accomplish this, the robots need to be able to navigate back to their positions at a given time. The robots will do so by taking the shortest straight-line path to their destination. If they encounter an obstacle, they will stop until the obstacle moves. If the other obstacle is a robot, one robot will move first, clearing the path for the other robot. If devices get stuck, an operator can intervene to manually guide the robots safely to their destinations.

2.3 Show Control Subsystem

The show control software itself sits on top of the design software. Running the show is simply a matter of triggering cues. The show control window shows a list of the cues in order and identifies the currently running cue. Cues are triggered either directly from that window or by incoming messages over the network.

During the show, real-time feedback is provided through either the Simulator or a more conventional tabular view. In this context, the simulation window simply visualizes feedback from the robots. Operators can manually take control of any device at any time, allowing them both to provide expressive live interactions as well as to prevent accidents. The show control system also allows operators to remotely disable any and all devices should a dangerous situation arise.

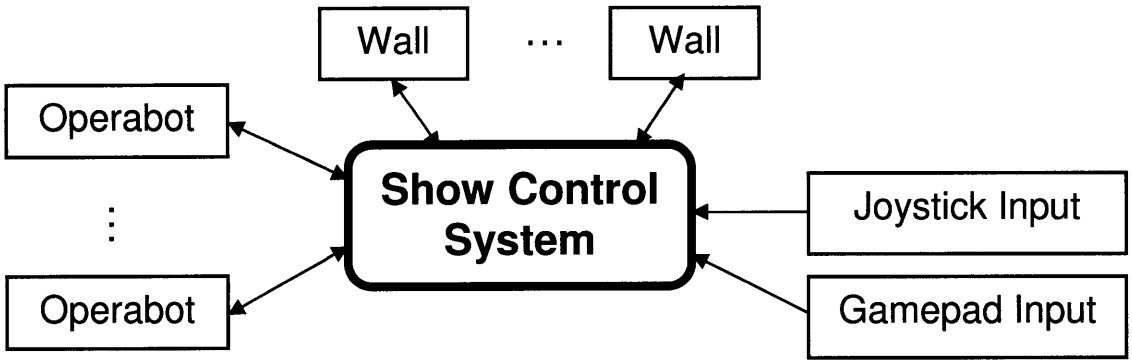

Figure 5. High-level overview of the show control subsystem

The show control also establishes a protocol for communicating with devices. The protocol draws inspiration from the Architecture for Control Networks (ACN) protocol [10]. When a device comes online, it immediately tries to connect to a known show control server via TCP. If the connection fails, it tries repeatedly to reconnect every short time interval. Once a TCP connection is established, the device offers the show control server a description of the available axes and the device's universal unique ID (UUID). The server responds with a request to join a multicast group. Finally, the client device accepts, joins the multicast group, and listens for any UDP messages.

UDP multicast allows the transmission of UDP messages to any clients subscribing to that address and port. Multicast groups are important in this application because they allow multiple devices to "snoop" on the traffic related to individual devices. Since there are multiple operators, it is desirable to have each at a separate computer. This requires each operator to have a pared-down version of the show control software and the Simulator. Instead of communicating directly with the server and creating more traffic, these thin clients can simply listen to the traffic across the multicast groups. In this manner, the displays can update using the data sent to and from the device without any extra overhead.

Similarly, other pieces of software can interact with these multicast groups. Audio localization software that positions notional sound sources in space might want to

tie sounds to the location of a device. By subscribing to the device's multicast group, the localization software can receive notifications about the robot's current position.

3. Implementation and Results

The show design and control application is written entirely in Java. This choice is somewhat controversial. The software is a performance critical system being used for real-time communication with a large number of client devices. Standard Java is notably not suited to real-time uses. Garbage collection can bring the application to a halt for seconds at a time, interrupting precise timing-related code. The *Real-Time Specification for Java* (RTSJ) attempts to address these issues by introducing uninterruptible threads and other tools [11].

Java was chosen first for its ease of use and second for its relatively good performance. The project's relatively short timeline and the desire for easy cross-platform portability make Java a suitable choice. C++ would have been a more appropriate choice with regard to real-time performance. Writing the application in C++, however, would likely have led to coding overhead, slower development iterations, and less rapid UI development. Java's standard libraries and Swing provide a convenient starting point. A similar setup in C++ would require a number of complicated external libraries such as Boost and wxWidgets. These libraries are admittedly more powerful and flexible than similar built-in libraries in Java but are more difficult to use at the same time.

The application is compatible with Java 6. It was developed in the Eclipse IDE, version 3.5 on a quad-core Apple Mac Pro. The application also augments the Java standard library with some additional libraries. JXLayer enhances Swing's windowing system with a pane that has its own glass pane and receives mouse notifications regardless of the listeners installed on child components. SwingX provides a set of useful Swing components such as a tree table. JInput adds support for controllers other than keyboards and mice such as joysticks and gamepads. Java does not have built-in support for 3D graphics. The Simulator uses the third-party

library Java Monkey Engine (jME) for 3D graphics in preference to a lower-level library like JOGL or LWJGL.

3.1 Design Patterns

The implementation of the show design and control software frequently relies on common design patterns to create reusable and familiar structures for interaction. As well, a few patterns were designed specifically for the show design and control software, such as the Property Pattern and the Identifier Pattern.

3.1.1 Singleton Pattern

The Singleton Pattern is a convenient method for creating a singular instance of a class that is globally accessible. A naïve solution in Java might simply create a class with only static methods and members. This style of singleton, however, is not adaptable. Turning an entirely static, singleton class into something instantiable requires a complete overhaul of the singleton class as well as any client code that interacts with it.

The Singleton Pattern creates easily adaptable singletons by using a private constructor and a static accessor method. Since no client can reach the private constructor, the singleton class itself constructs a single static instance and provides a static getter for that instance. If the singleton eventually needs to be changed to support multiple instances, one simply removes the static method and member. All the client code that interacted with the singleton can remain essentially the same. The client code simply needs to invoke the same commands on a passed-in reference rather than the singleton reference.

3.1.2 Observer Pattern

The Observer Pattern is an object notification methodology. It allows for loosely connected objects that respond to updates instead of querying for updates. Java's Swing toolkit uses the Observer Pattern extensively. If a component wants to respond to mouse click events, for example, it simply registers a mouse listener that performs the requisite actions. Whenever a mouse click occurs on the component, Swing invokes that component's registered mouse listeners.

3.1.3 Property Pattern

The Property Pattern is a custom pattern created explicitly for this application. It relates closely to the Observer Pattern and draws inspiration from JavaBeans. While the Observer Pattern makes writing listener clients easy, making a class observable is tedious and involves a lot of nearly identical code. The Property Pattern abstracts away manual listener management and event notification.

Properties are wrappers around what would normally be member variables. A property table maps property names to property values. It also manages any listeners. Whenever a client sets a property's value, the property table automatically fires a `PropertyChangedEvent` to any registered `PropertyChangeListeners`.

The Property Pattern also allows for transactional property edits. A property stores not only its current value but also the value of its last checkpoint. Edits can be committed, creating a checkpoint at the new value, or reverted to the previous value. This allows easy integration with undo-redo functionality. These edits can automatically yield structures like Swing's `UndoableEdit`.

The Property Pattern also trivializes generalized input and output (I/O). Writing an object to disk, for instance, is a matter of simply writing the class name and the properties.

Objects that apply the Property Pattern gain all the benefits of the Observer Pattern without the implementation overhead, as well as additional benefits such as atomic edits and automated I/O.

3.1.4 Visitor Pattern

The Visitor Pattern has its roots in functional programming. Typical object-oriented styles construe functions as methods belonging to an object and its related descendents. The Visitor Pattern views functions as methods that operate on and across possibly disparate sets of objects.

In a traditional object-oriented approach, adding output support to a varied set of objects would require adding a method to each object. Each object would then be responsible for writing itself. From a conceptual point of view, however, the "write" method is not really a function of the object—it is an operation across the object. A second pass might yield a "writer" class that knows how to write the relevant types of objects. However, this approach is not type-aware. That is, if the objects to be written are cast to a common base class, the writer would have no way to determine how to write that object without repeated invocations of Java's `instanceof` operator.

The Visitor Pattern solves the method dispatch problem above by taking advantage of inheritance. Each "visitable" client object must implement a method that accepts a visitor (function) object. Java invokes the most specific override of the acceptor method. Thus, inside that method, the `this` object will have the correct, specific

underlying type. The acceptor method simply invokes the visitor's function with the correctly typed `this` object.

3.1.5 Factory Pattern

The Factory Pattern is used to encapsulate the steps related to object creation. Object creation sometimes involves additional steps beyond the invocation of the constructor. The factory pattern can abstract these steps into a single invocation, promoting code reuse and adaptability. As well, it is impossible to change the type of an object in the constructor. Thus, if the type of an instance depends on the values or type of its constructor arguments, the Factory Pattern is a necessity. Often, the Factory Pattern is used in conjunction with other patterns such as the Singleton Pattern (global access) or the Visitor Pattern (type-awareness).

3.1.6 Identifier Pattern

The Identifier Pattern is another pattern peculiar to the show design and control software that addresses an issue that occurred multiple times during the development process. Sometimes it is useful to generate unique keys for instances of some type. These keys make the instances readily storable in a centralized location via a hash map. When paired with a centralized table of instances, the identifier can also provide a shortcut method to simply retrieve its associated object from such a table. As well, the object can override `Object`'s `equals` method to simply compare identifiers. The identifier's implementation can be anything ranging from guaranteed-unique integer to a full-fledged UUID.

3.1.7 Curiously Recurring Template Pattern (CRTP)

The Curiously Recurring Template Pattern (CRTP) is an odd pattern that has its genesis in C++ template programming. In this pattern, a derived class extends a generic base class parameterized by the derived class. While it is more powerful when coupled with C++'s templates, it is still very useful with Java's generics.

Because of Java's type erasure, it is normally impossible to infer the type parameters that were used to instantiate a generic class: `List<Integer>` and `List<Float>` become indistinguishable at runtime. When a derived class extends an explicit instantiation of a generic base class, the type is not erased. Using Java reflection, the base class can determine the class of the parameterized types. Thus, the CRTP is useful for circumventing generic type erasure and giving the generic base class knowledge of its derived class.

3.2 Framework

Neither Java standard libraries nor Swing include a comprehensive document framework. As such, the application uses a custom-built framework. This Document Framework automates much of the boilerplate code involved in building a Model-View-Controller-based user interface. The API relies on inheritance, reflection, and multiple design patterns to allow rapid development of user–interface-friendly objects.

3.2.1 Documents and DocumentElements

The two main, user-facing components in the Document Framework are `Document`s and `DocumentElement`s. `Document`s are primarily responsible for maintaining a repository of their associated `DocumentElement`s. For instance,

`SequenceDocument` is a document that arranges a number of `Clip`s and `Track`s. Both `Clip` and `Track` are types of `DocumentElement`s.

While the types of `Document`s are relatively fixed, the number of functions across `Document`s is always growing. `Document`s implement the Visitor Pattern to accommodate additions that span different types of `Document`s. As well, functionality such as XML I/O is not really part of a `Document`, so the Visitor Pattern abstracts this functionality away from the `Document` class and into a visitor class.

Unlike a conventional notion of a document, a `Document` does not necessarily represent a file on disk. In the current show file format, the entire show and all the `Document`s within are stored in one file. This is not a requirement of the system, however. It might be convenient to store each `SequenceDocument` in its own file, for instance. The framework is flexible enough to accommodate such design decisions.

`Document`s are responsible for managing their own undo-redo stack. Rather than using a global undo manager, changes localized to a document are independently undoable. This is standard behavior for any software that supports multiple documents. If a user has multiple documents open, undoing changes in one should not affect the others.

A `DocumentID` uniquely identifies each `Document`. `DocumentID`s adhere to the Identifier Pattern and are implemented using a randomly generated UUID. A central repository called the `DocumentManager` maintains a globally accessible mapping of `DocumentID`s to their respective `Document`s. The `DocumentManager` uses the Singleton Pattern. Client code rarely needs to interact with the `DocumentManager`. When a `Document` is created, it automatically adds itself to the `DocumentManager`. `DocumentID`s also provide a getter method that simply retrieves the associated `Document` from the manager.

DocumentElements are component entities of the overall Document. The individual tracks in a sequence are elements of the SequenceDocument, for example. DocumentElements apply the Curiously Recurring Template Pattern, so a DocumentElement is parameterized by a class derived from a DocumentElement. Clients create new entities by extending DocumentElement. A DocumentElement belongs to exactly one Document.

DocumentElements implement both the Visitor Pattern and the Property Pattern. Both are necessary for automating processes like undo-redo and I/O. As well, the Property Pattern encourages clients to use the Observer Pattern to structure communication among other DocumentElements and view components.

ElementIDs uniquely identify every DocumentElement. ElementIDs use the Identifier Pattern. An ElementID consists of a DocumentID; the class of the DocumentElement identified; and a monotonically increasing number incremented with each new ElementID of the same type within the same Document. Documents are responsible for storing the next highest integer for each type of DocumentElement stored within. Like DocumentIDs, ElementIDs provide a getter method that retrieves the uniquely identified DocumentElement from the Document.

3.2.2 Views and ViewElements

Views are a specific type of DocumentElement. In the Model-View-Controller Pattern, the view refers to the visual representation. In this case, the View manages information about the UI-specific details of a given view of a Document. A Document can be presented in multiple Views. View-specific elements, however, are unique to the specific view. For instance, in the SequenceView for a SequenceDocument, a clip selection exists only within the View in which the

selection occurred. Thus, the user could have two views of the same `Document` with two different selections.

`Views` allow for hierarchical nesting, mimicking the way that UI components may contain other components. The root `View` is likely displayed in its own window (although this is not strictly required). This nesting allows child `Views` to retrieve properties from the parent view.

`Views` offer a replacement for Swing's limited focus subsystem. Much of Swing's functionality with regards to keyboard accelerators and `Actions` relies on the focus subsystem to work. However, custom components do not always interact nicely with the focus subsystem. A custom `KeyEventDispatcher` notifies the active view instead of the currently focused UI component. The `View` can decide how to respond to key events since concepts like selection and focus are handled by the `View` and not Swing. If a `View` does not have an `Action` bound to a given key press, it refers the event to its parent `View`, if available. Otherwise, the keystroke will have no effect.

While `View` extends `DocumentElement`, it also shares commonalities with `Document`. Like `Document`, each `View` maintains a table of entities analogously called `ViewElements`. `Views` also keep track of their own set of monotonically increasing ID numbers per `ViewElement` class as `Document` does with `DocumentElements`. While `Documents` manage edits and the undo-redo history, `Views` direct edits to the proper `Document`. Edits are forwarded to the `Document` associated with the root `View`, even if the edit is performed in a nested `View` displaying a different `Document`.

`ViewElements`, like `DocumentElements`, are the component entities of a `View`. Just as `Views` provide one perspective on a `Document`, `ViewElements` provide a perspective on a `DocumentElement`. There are generally two kinds of

`ViewElement`s: non-visual elements that model various portions of the view and visual Swing components. `ViewElementID`s are used to uniquely identify `ViewElement`s. `ViewElementID` extends `ElementID` and specifies the `View` that owns the identified `ViewElement`. Otherwise, `ViewElementID`s are identical to `ElementID`s.

3.3 Sequence Editor

The Sequence Editor is the user-facing tool for developing sequences, the data and routing portion of a cue. It is the primary component of the design subsystem, a comprehensive user interface for interacting with the tracks and automation signals. The Sequence Editor is implemented using the Document Framework and Swing. The UI is designed around a timeline and modeled after mainstream audio sequencers and non-linear video-editing tools.

3.3.1 Sequences and Tracks

The Sequence Editor operates on a `SequenceDocument`. `SequenceDocument` extends `Document`. A `SequenceDocument` holds a single instance of a `Sequence`, but otherwise has no additional functionality beyond a `Document`.

A `Sequence` is a `DocumentElement` responsible for managing the state of the `SequenceDocument`. It maintains the current playback time and the root of the track hierarchy as properties. Since there is only one `Sequence` per `SequenceDocument`, `Sequence` could easily have been written within `SequenceDocument` itself. `Document`s do not currently use the Property Pattern (although there is no reason why they could not). Thus, it seemed slightly more appropriate to use a `DocumentElement`.

Tracks fall into three categories: standard, composite, and group. Standard tracks arrange clips of data linearly along the timeline. Composite tracks combine the value of their children—either standard or composite tracks—using some blending operation such as addition or multiplication. Group tracks simply provide a grouping with no output functionality.

All three categories share a common base class, `AbstractTrack`, which extends `DocumentElement`. `AbstractTrack`s have three properties: a name, a height in pixels, and a color. These properties affect the visual display of the track and its contents.

`TrackGroup` implements the group track category. It has a list of `AbstractTrack` children and provides methods to add and remove tracks. Another abstract type, `BasicTrack`, subclasses `AbstractTrack`. `BasicTrack` clarifies the primary distinction between standard and composite tracks and track groups: standard and composite tracks have an output value. `BasicTrack`s therefore add output-related functionality like metering and value-to-pixel scaling. Each `BasicTrack` is associated with a given `OutputAxis` that represents the output target on some device. `CompositeTrack` covers the composite track category by adding a list of `BasicTrack` children and a blending mode property. `Track` accounts for the standard track category. It adds a list of `Clip`s.

3.3.2 Clips and Data

`Clip`s are the data currency of `Track`s. A `Clip` is a `DocumentElement` with an absolute time within the track and duration. `Clip`s also provide a windowed view of the underlying data. `Clip`s have properties for start time, and start and end "crops" which define both the duration and the viewed portion of the underlying data.

The `TrackData` interface encapsulates the data viewed by `Clips`. This data eventually becomes the output value of a `BasicTrack`. `TrackData` refers to anything that has a value in time. Implementations simply fill in the method `getValueAtTime`. `Curves` are one variety of `TrackData`. `JoystickData`, with its value in time based on the current value of an axis of the joystick, is another type of `TrackData`. Other potential `TrackData` variants include live input received over the network and short "scripts" that do some calculation to create a value.

3.3.3 Curves

`Curves` are the most complex type of `TrackData` implemented. They are defined in terms of a flexible key framing model using third-order Bézier curves with some extensions. A `Curve` is a `DocumentElement` with a special set of `Keys` called a `KeySet` and additional properties like an end behavior that controls how the curve is defined for times beyond its last `Key`.

A third-order Bézier curve is defined by two end points each paired with a control point. Bézier curves are often defined in terms of a parametric equation over the range $t \in [0.0, 1.0]$:

$$C(t) = \sum_{i=0}^{n} B_{i,n}(t) \cdot \mathbf{P}_i$$

Where $B_{i,n}$ is the i^{th} Bernstein polynomial of order n, and $\boldsymbol{P}_i$ is the i^{th} point. $\boldsymbol{P}_0$ and $\boldsymbol{P}_3$ are the end points, while $\boldsymbol{P}_1$ is the control point for point $\boldsymbol{P}_0$ and $\boldsymbol{P}_2$ is the control point for point $\boldsymbol{P}_3$. The same equation can be explicitly rewritten as a third-order parametric curve:

$$C(t) = (1 - t^3)\boldsymbol{P}_0 + 3t(1 - t^2)\boldsymbol{P}_1 + 3t^2(1 - t)\boldsymbol{P}_2 + t^3\boldsymbol{P}_3$$

The formula unfortunately does not provide much insight into how a curve will look given its four points. It is helpful to have an intuitive picture of Bézier curves when

working with them in any capacity. The curve defined by these points always passes through the end points. The two control points provide shape within the end constraints. The curve initially moves tangent to the line between the end point and its associated control point. The distance of the control point from its end point affects how heavily that control point affects the curve.

A third-order Bézier curve can define any traditional polynomial up to third-order. It can also define lower-order curves such as second-order curves and lines. The type of curve that results depends entirely on the position of the two internal points. If the points both lie on the line between $\boldsymbol{P}_0$ and $\boldsymbol{P}_3$, the result is simply a line. If only one point lies on the line, it will start out linear but gradually transition as the other control point exerts more weight on the curve.

In the show design and control software, a `Curve` is comprised of third-order Bézier curves attached end-to-end. These segments are defined between adjacent keys. A key is an end point with "in" and "out" control points. The "in" control point affects the preceding segment while the "out" control point affects the next.

Keys are implemented as the class `Key`, which extends `DocumentElement`. `Key` has properties that define its location in time; its value; "in" and "out" control point locations; and "in" and "out" interpolation types. The interpolation types force constraints on the control points. For instance, the linear interpolation type guarantees that the control point always lies on the line between its key and the previous or next key (depending on whether it is "in" or "out" interpolation, respectively). Other available interpolation types include step, hold, and smooth. Step and hold interpolations are only available for out interpolations. The value over a step-interpolated curve segment is simply the value of the next key. Similarly, the value over a hold-interpolated segment is the value of the current key. Smooth interpolation produces a "smooth" third-order polynomial by forcing the

control point to the horizontal axis of the key, situated in time halfway between the previous or next key.

`TrackData`'s `getValueAtTime` method requires `Curve` to return its value at a given moment in time. The mathematical formulation above is unfortunately not defined in terms of time. The curve needs to be interpolated to find the value of the curve at a particular moment in time. There are two different methods of interpolation: naïve traversal of the parametric equation using a small step size, or an adaptive, recursive procedure called De Casteljau's algorithm.

Iterating through the parametric equation is easy to implement. Unfortunately, it is computationally intensive for small iteration steps. On the other hand, large iteration steps fail to capture large curvature changes accurately. The iterative method consequently produces a linearly spaced set of points. If the step size is small, a linear Bézier segment may be defined using a large number of points when only two are necessary.

De Casteljau's algorithm recursively subdivides a given Bézier curve into two new Bézier curves that produce an identical representation. It stops subdividing when it reaches a certain flatness threshold. Recall that a Bézier curve can be linear when the control points are in line with the end points. Thus, the base case for the recursion is when a Bézier curve is within some threshold of linear. The resulting interpolation, then, is the set of end points of all the Bézier curves traversed. This produces an interpolation with more detail in particularly curved areas and less detail in nearly linear regions. Compared to the iterative algorithm, De Casteljau's algorithm produces higher quality, adaptive renditions with less computation and less memory.

Both algorithms are implemented for reference on top of the class Interpolation. `ParametricInterpolation` uses the iterative method while

`AdaptiveInterpolation` uses De Casteljau's algorithm. Currently, only the `AdaptiveInterpolation` is used in the implementation. `ParametricInterpolation` is provided simply for reference.

3.3.4 Axes and Devices

Axes represent the physical reality of the external robots connected to the show control system. Generally speaking, a device has some degrees of control. A single light channel on the Operabots is one such degree. The light accepts intensity values between zero and one. These values dictate the output brightness of the light source. An `Axis` is analogous to such a degree of control.

An `Axis` can come with a variety of additional restrictions. It can be bounded, as with a light, such that the range of possible values is constrained within an acceptable range. An `Axis` can also refer to an input, an output, or both. The light channel is an output axis in that the robot sends a value to that axis. Queryable components such as remaining battery life or data from a distance sensor are inputs. Different types of axes are realized by extending the `Axis` base class. Existing subclasses include `InputAxis`, `OutputAxis`, `BoundedInputAxis`, and `BoundedOutputAxis`.

As the need arises to model more complex devices, different types of axes may become necessary. It might be useful to represent the same degrees in two different manners. Consider the drive system on the Operabots. For live input situations, it is convenient to imagine the drive as controlled by three different output parameters: X speed, Y speed, and rotation speed. This is natural for using a controller like a joystick to drive a robot. However, when creating complex choreography, this representation is unhelpful—it does not contain any notion of the absolute position on stage of the robot.

A more sophisticated representation would include absolute X and Y positions in addition to the other three parameters. These five axes are not independent. For an Operabot to reach a specific location on stage by some certain time, the robot's speeds need to be adjusted. Similarly, for the robots to reach the same location at a slower speed, it will arrive at that point later in time. The current implementation does not account for these sorts of linked axes. With changes like linked axes in mind, the design has been left as general as possible so as to make future additions straightforward.

`Device`s establish logical groupings and names for axes. An `Axis` is a `Device` with a name. Currently, the only other type of `Device` is `DeviceGroup`. A `DeviceGroup` has a list of `Device` children. A `DeviceGroup` therefore allows for a nested hierarchy of `Device`s.

Groupings may be aesthetic or functional (or somewhere in between). It is convenient to group a large number of axes into various subdevices simply for organization. Consider modeling the Operabot. The Operabot is a device made up of subdevices so it is a `DeviceGroup`. There are three logical subsystems: drive, articulation, and lighting. The root `DeviceGroup` therefore has three child `DeviceGroup`s. The drive `DeviceGroup` has `BoundedOutputAxis` instances for X, Y, and rotation speed bounded by the range [-1.0, 1.0]. The articulation `DeviceGroup` has two `BoundedOutputAxis` instances for elevation position, bounded by [0.0, 1.0] and head position bounded by [-1.0, 1.0]. Finally, the lighting `DeviceGroup` has eleven `BoundedOutputAxis` instances bounded in the range [0.0, 1.0], one axis for each lighting channel.

3.3.5 Sequence and Track UI

The user interface for the Sequence Editor revolves around the `SequenceView`. The `SequenceView` is the data model relevant to a specific view of a `Sequence`. In this

sense, the same `Sequence` can be shown in two different ways. Both will share features with the `SequenceDocument` but parameters such as the clip selection in the `SequenceView` are unique to that specific visual realization.

`SequenceView` defines a number of member variables necessary for visualization of and user interaction with `Sequence`s. It has a time scale (appropriately named `TimeScale`) for the view that converts time in seconds to pixels on screen. It keeps track of the current clip selection using a generic class called `ElementSelection`. `ElementSelection` leverages the Document Framework, providing an easy way to manage selection of `DocumentElement`s. `SequenceView` also stores the currently focused track. Track focus is necessary for pasting clips, for instance. Other properties and values managed by the `SequenceView` include the selection cursor's time; a snap mode for determining how (or if) to quantize time-based interactions to the grid; and mappings from `AbstractTrack`s and `Clip`s to their associated UI panes.

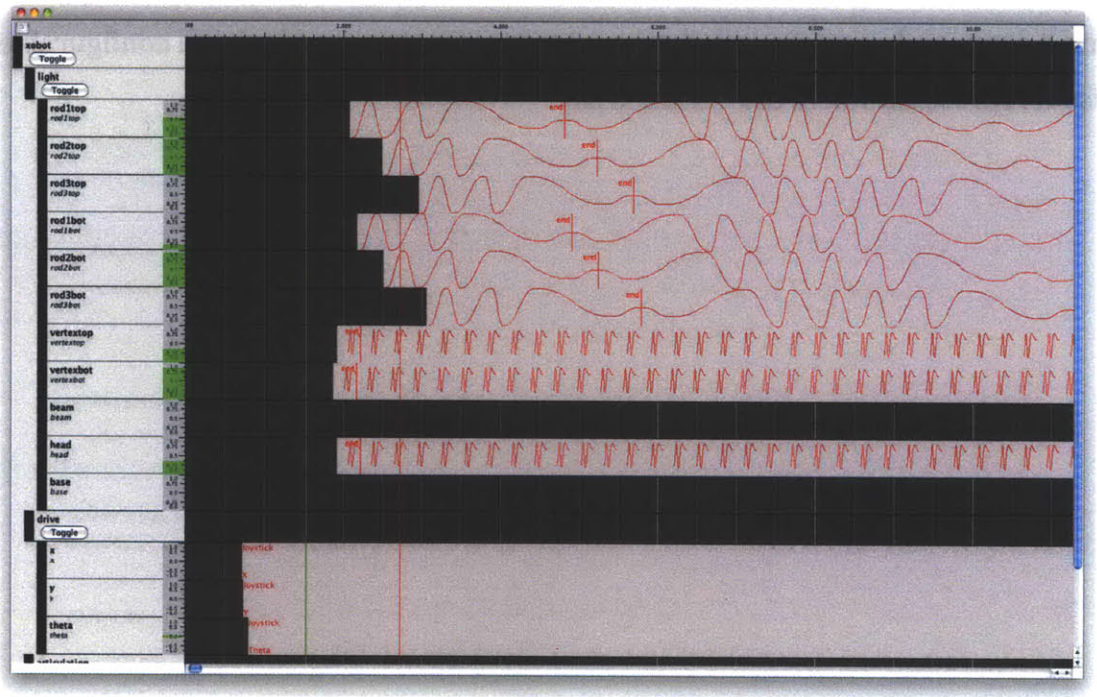

Figure 6. The Sequence Editor in action, playing back a demo sequence for an Operabot

`SequencePane` is a `JComponent` that contains the entire contents of the window

(Figure 6). It uses a `JScrollPane` to create a scrollable viewport of all the tracks. Tracks are represented with an "info" pane which displays information about the track such as its name and color. The info pane also includes a vertical ruler, which shows the value range of the `BoundedOutputAxis` associated with the `Track`. To the right of the info pane is the "lane" pane which exists on the timeline. The lane is the vertical and horizontal space which the track encompasses. `Clip`s are displayed in the lane for `Track`s and the composite output value is shown in the lane for `CompositeTrack`s. `TrackGroup`s simply show the underlying time grid.

Time rulers play an important role in displaying the sequence. Along the top of the `SequencePane` there is a ruler (`TimeRuler`) that displays the current time at scale according to the `SequenceView`'s `TimeScale`. Clicking or dragging on the ruler moves the playback position to the selected time. Clicking while holding down the meta key zooms in around the clicked time, while additionally holding shift zooms out. `Lane` panes display a time grid in the background that lines up with the major grid ticks on the `TimeRuler`. This shared ruler-like functionality was abstracted into a common base class called `AbstractTimeGridPane` which is a configurable visual representation of the `TimeScale`.

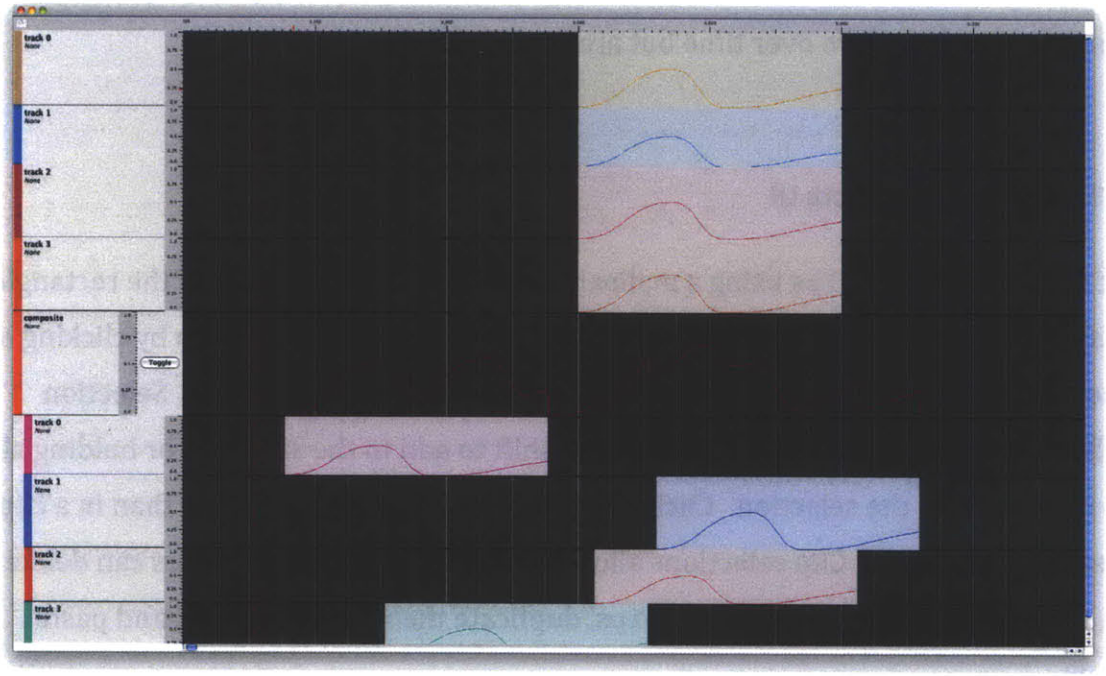

Figure 7. The Sequence Editor with colored tracks and a composite track that adds its children

`Lane`s are implemented using two `ViewElement`s: a non-visual abstraction called `Lane` and a `JComponent` called a `LanePane`. The non-visual `Lane` encapsulates the idea of not simply the visual lane, but also the info pane with which that lane is connected. `Lane`s are created using an instance of `LaneFactory`, a visitor which leverages type-awareness onto the Factory Pattern to construct the proper type of `Lane` depending on the type of track. `TrackLane`, `CompositeTrackLane`, and `TrackGroupLane` subclass `Lane`. These `Lane`s are associated with `TrackLanePane`, `CompositeTrackLanePane`, and their common base class `LanePane`, respectively.

These type distinctions mean that `Track`s, `CompositeTrack`s, and `TrackGroup`s have different view and controller aspects. For instance, both `CompositeTrack`s and `TrackGroup`s have a toggle button that shows or hides its list of children. Only `Track`s can display `ClipPane`s, and drag-and-drop only works on `TrackLanePane`s. Moreover, selection, the selection cursor, and track focus only

apply to `TrackLanePane`s. `CompositeTrackLanePane`s simply display the composite value curve over time but are not modifiable.

3.3.6 Clip and TrackData UI

`ClipPane`s depict `Clip`s using a malleable rectangle. The contents of the rectangle display the `TrackData` viewed by the `Clip`. The user can select a clip by clicking on the `ClipPane` or dragging a selection across a swathe of the timeline. Selection follows common conventions like holding shift to add to the selection or holding alt to remove from the selection. Clicking on the `TrackLanePane` rather than in a clip clears the selection. Clip selections allow for actions across clips. Users can delete `ClipPane`s and their underlying `Clip`s, duplicate them, and cut, copy, and paste them.

`ClipPane`s support a number of modes of user interaction. Users can drag-and-drop `ClipPane`s from Track to Track. The entire `Clip` selection for the `SequenceView` can be moved at once, maintaining the relative spacing between clips in time as they move. The spacing across tracks is similarly preserved, even when dragging between different tracks. Clicking and dragging a `ClipPane` near either edge manipulates the start or end crop for the `Clip`, respectively, revealing or obscuring the underlying `TrackData`. Edge-editing a clip selection with more than one clip moves the edges on the entire selection. The three current modes of interaction are distinguished by where the user clicks to begin the operation. Different cursors provide visual feedback when the user places the mouse over one of these three hotspots. In the future, additional modes may be added to support functionality such as stretching the underlying data in time or value.

The Document Framework makes supporting undo-redo painless. All the actions across clips, as well as the modal click-and-drag operations, are undoable. The modal operations simply update the model as if there was no concern for undo-

redo. When the operation ends, the program simply commits or reverts the edit through the Property Pattern's interface with one method invokation. Committing the change returns a neatly packaged `UndoableEdit` object that can be added to the `SequenceView`'s (and hence the `SequenceDocument`'s) undo stack.

`TrackData` is shown within the `ClipPane`'s rectangle. To facilitate working with different variants of `TrackData`, a `ClipPane` requires a `TrackDataRenderer` component. The `TrackDataRenderer` is responsible for displaying the contents of the `TrackData` and providing means for editing it as well. `CurvePane`, the most complex `TrackDataRenderer`, is discussed in depth in the next section. `JoystickRenderer` is a conceptually simpler example. `JoystickRenderer` is the visual representation of `JoystickData`. For visual feedback, it simply displays a label in the top left corner that reads "Joystick" and a label in the bottom left corner that shows the axis ("X", "Y", or "Theta") from which the value derives. The user can edit the chosen joystick axis by double-clicking, which brings up a dialog box that asks the user to choose which axis on the controller to use.

3.3.7 Curve UI

The `CurveView` controls the state of the `Curve` UI much as the `SequenceView` manages the `Sequence` UI. The `CurveView` interfaces with the `SequenceView` through the `TrackDataRenderer` class used by `ClipPane`. `CurvePane` extends `TrackDataRenderer` and draws the `Curve`. `CurveEditPane` is the editor layer of the `TrackDataRenderer`, allowing users to manipulate `Key`s, control points, and interpolation types for the represented `Curve`.

CurvePane can draw the Curve using different methods. To support different rendering schemes, the CurvePane uses the CurveRenderer interface. This interface requires an implementation to fill in the render method that returns a Java2D Shape. This Shape depicts the Curve without taking into account the end

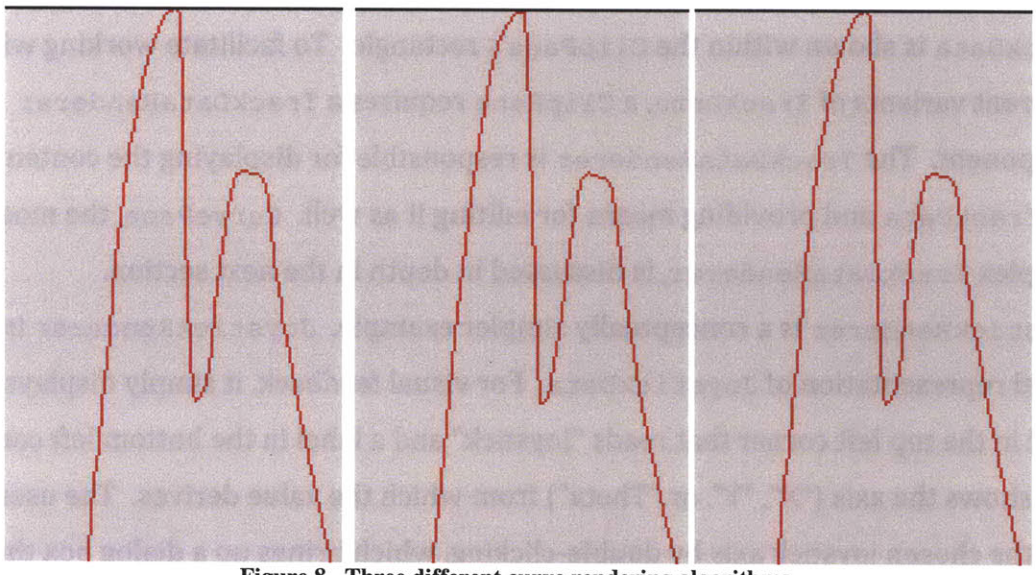

Figure 8. Three different curve rendering algorithms
(from left to right: pixel, interpolation, and Bezier rendering methods)
Although nearly indistinguishable visually, the different renderers have very different performance implications.

behavior. The CurvePane draws the Shape and repeats it with various transformations to realize the correct end behavior.

There are currently three different implementations of CurveRenderers: PixelCurveRenderer, InterpolationCurveRenderer, and BezierCurveRenderer (Figure 8). PixelCurveRenderer iterates pixel-by-pixel horizontally, asking the Curve for the value at that pixel's time (calculated using the TimeScale). InterpolationCurveRenderer takes a slightly more intelligent approach. It iterates through the rendered interpolations stored in the Curve. It simply draws a line between each point in the interpolation. Coupled with AdaptiveInterpolations, this method can be very efficient. Finally, BezierCurveRenderer uses Java2D's native Bézier curve support in GeneralPath.

All three rendering methods produce nearly identical results, although efficiency varies. `InterpolationCurveRenderer` and `BezierCurveRenderer` are more efficient than `PixelCurveRenderer`. `BezierCurveRenderer` is more closely integrated with the hardware, so it is generally faster. However, `InterpolationCurveRenderer` could be more efficient depending on the flatness threshold used in the `AdaptiveInterpolation`. `InterpolationCurveRenderer` could draw very fast but less accurate renditions of the `Curve`. At this point only `BezierCurveRenderer` is used with the expectation that it will provide better speed due to hardware acceleration. The other two implementations are provided as a reference to previous methods.

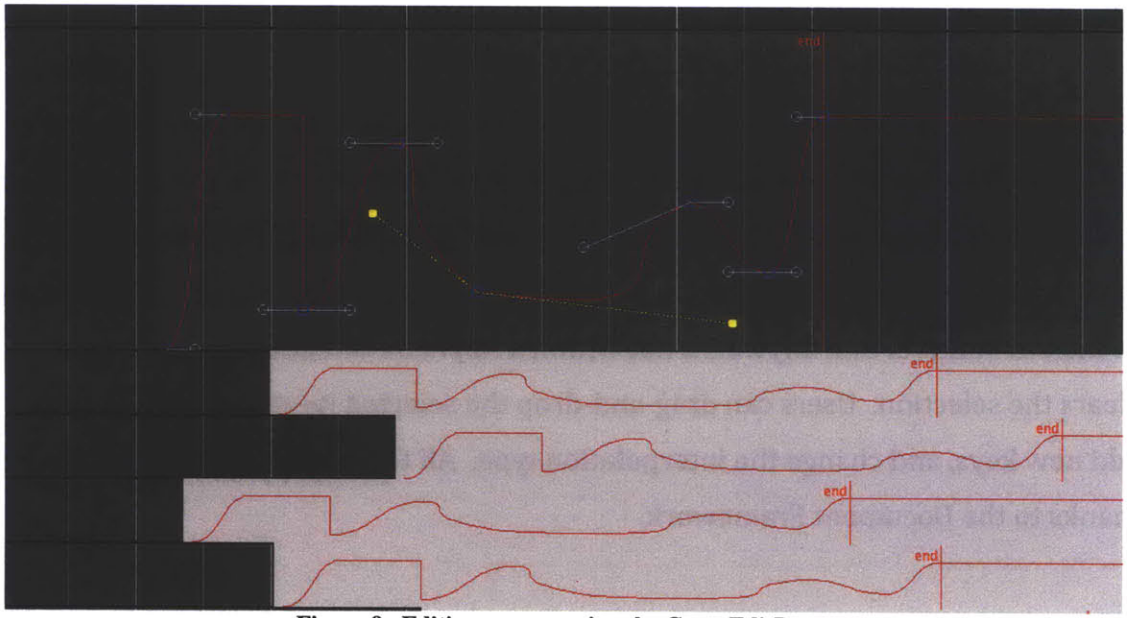

Figure 9. Editing a curve using the CurveEditPane

The `CurveEditPane` is the heart of the `Curve` UI, allowing the user to manipulate the underlying `Curve` (Figure 9). Double-clicking on the `CurvePane` overlays the `CurveEditPane`. `KeyPane`s visualize the `Key`s using a simple colored square. When the `KeyPane` is selected, the square is filled. Otherwise, it is hollow. `ControlPointPane`s similarly display the control points for the `Key` but with a circle instead of a square. A dashed line visually attaches the `ControlPointPane`s to their parent `KeyPane`s.

Both KeyPanes and ControlPointPanes share functionality. They both are visually similar, represented by colored, primitive shapes. And both can be clicked and dragged around the CurveEditPane. As such, they share a base class that encapsulates this common functionality called HandlePane.

While both types of HandlePane have much in common, the two are also functionally different. ControlPointPanes, unlike KeyPanes, are generally not editable. Interpolation types enforce constraints on the locations of the control points with the exception of "free" Bézier interpolation. Thus, only unconstrained control points are editable. As well, right clicking on a ControlPointPane brings up a different contextual menu than right clicking on a KeyPane.

KeyPanes support operations similar to ClipPanes. The user can select KeyPanes (and editable ControlPointPanes) by clicking on them or dragging a selection box around them. As with ClipPanes, selections can be modified by holding down the shift or alt key to add or remove KeyPanes from the selection, respectively. Clicking in the CurveEditPane anywhere not within a KeyPane or ControlPointPane clears the selection. Users can drag-and-drop the selected KeyPanes, delete them, add new keys, and change the interpolation type. All these operations are undoable, thanks to the Document Framework.

3.3.8 File I/O

The data in a show, including entire SequenceDocuments and CurveDocuments, are stored on disk using an XML-based file format. XML is convenient because it is human-readable, making it easy both to edit manually and debug.

Reading and writing of show XML files leverages the Document Framework's use of the Visitor Pattern and Property Pattern. The Visitor Pattern allows for pluggable

writer objects capable of traversing any `Document` or `DocumentElement` in the system. In the simplest case, outputting a `DocumentElement` requires no extra code beyond writing the `DocumentElement` itself. Any "primitive" `DocumentElement` can be reconstructed completely from its class name and a list of its properties and associated values. Here, primitive refers to a `DocumentElement` that has only transient data members and persistent properties. Writing a property simply means writing the property name, its value, and the class name of the value.

Writing a `Document` to disk is a matter of first traversing the tree of reachable `DocumentElement`s and collecting them into a set. In the process of working on `Document`s, a `DocumentElement` may cease to be referenced yet may not be removed from the `Document` itself. This collection step prunes the dead objects so they are not written to disk. `CurveDocumentUsageVisitor` and `SequenceDocumentUsageVisitor` both walk their respective element trees for this purpose. Next, each active `DocumentElement` is labeled with a monotonically increasing integer-as-string XML ID. A map maintains the relation between `ElementID`s and their respective XML IDs.

Finally, an implementation of `DocumentElementXMLVisitor` such as `CurveDocumentXMLVisitor` or `SequenceDocumentXMLVisitor` walks the set of active elements with the `ElementID`-to-XML–ID mapping. Each element is written to the file along with its XML ID. If a `DocumentElement` references another element, instead of nesting elements, the visitor writes an indirection that points to the referenced element's XML ID. In this manner, the output process flattens the deeply nested `DocumentElement` tree into a list.

Reading the XML is substantially more complex than writing it. First, one must choose an appropriate parser for the task. Java includes two XML parsers, each with substantially different ideologies. Java's DOM XML parser reads in an entire XML file at once and builds an in-memory tree for traversal. As such, the DOM parser is more

resource intensive and heavyweight in nature. Moreover, it is ill-suited for very large XML files that could take a long time to parse and interpret in two separate steps. Large XML files would lead to massive DOM trees, incurring a temporary memory hit.

On the other hand, Java's SAX parser walks the XML file, invoking callbacks as it encounters tags. SAX parses and interprets in one pass. This approach requires more work on the part of the client code, which usually must supply some sort of stack or context to maintain the current object hierarchy. While it is more difficult to work with, SAX has a much smaller memory footprint and should scale more gracefully with large XML files.

SAX is the appropriate choice in this case, for performance reasons alone. The show documents are very large, storing one-and-a-half hours of instructions for at least nine Operabots and the Walls. With SAX, however, comes added implementation complexity. Additional instrumentation is necessary to provide a context for the parser.

An `XMLEnvironment` interface represents the `Document`-level scope of the show XML file. It provides methods to register and lookup `Document`s encountered so far, and maintains a stack of `XMLContext`s. Each `XMLContext` maintains a stack of `DocumentElement`s in the midst of being initialized and a stack of XML tags in progress. It also keeps a map of XML IDs to `ElementID`s, the reverse of the mapping used by the `DocumentElementXMLVisitor`s. The `ElementID`s, however, are not the same as when saved: they are newly created. Thus, a freshly opened file's `ElementID`s are adjacent with monotonically increasing internal ID numbers. This process effectively clears the ID "holes" left by the unused elements that were not written to disk.

`DocumentElement`s are created as their XML tags are encountered and are added to the current `XMLContext`. However, it is possible (and likely) that a `DocumentElement` will reference an element that has not yet been created. In this case, because the element reference includes the class name of the referent, a `DocumentElement` is constructed, but not added to the stack. In either case, whenever a `DocumentElement` is created, its XML ID and resulting `ElementID` are added to the context's map.

The `ShowXMLReader` class uses the SAX `DefaultHandler` API. As `ShowXMLReader` encounter start and end tags, it simply delegates to an appropriate `XMLHandler` enumeration value. Each constant in the enumeration defines behavior for a tag used in the XML format. The specific enumeration constant can override the `start` method, which takes a SAX attributes list and an `XMLContext`, and `end` method, which takes only an `XMLContext`. If there is no enumeration constant appropriate for a given tag, a no-op `XMLHandler` constant is used.

3.4 Show Control

The show control portion of the application refers to the playback and server aspects. Playback is the process of reading track values and sending them to the client devices at a fixed time interval. This playback is analogous to audio or video playback in the sense that it takes a time-varying data source and reproduces it in real-time.

3.4.1 Playback

Playback is one of the most time-sensitive and performance-critical processes in the entire system. To present complex choreography, the timing should be accurate, fine-grained, and repeatable. Java's garbage collection and other slowdowns can

seriously hamper the playback thread's ability to perform reliably. As such, extra thought has gone into the performance characteristics of the playback mechanism.

A single timing thread triggers each time step of the playback mechanism and updates the global playback time. This `PlaybackDaemon`, which extends the custom timing thread `TimerThread`, uses a small thread pool to divvy up the task of evaluating the tracks at the current playback time. Each thread is passed a `TrackEvaluator` task with an `AbstractTrack` tree. The `TrackEvaluator` walks the tree using a `TrackValueVisitor`, collecting a list of `TrackValues` and their associated destinations. It then asks the server to send the gathered data as a large bundle to the target `Device` associated with that `AbstractTrack` tree's root.

Playback is the link between the show control and the design tools. To ensure proper device routing, the Sequence Editor's device-level `TrackGroups` must be associated with a specific client device connected to the server. A rudimentary implementation of this functionality called the Routing Window is available in the "Show" menu of the Sequence Editor. It contains a table of connected clients and allows the user to map a client to a specific set of tracks in the `Sequence`.

3.4.2 Device Protocol

The Server Device Protocol dictates how client devices connect to the server and what that process entails. This protocol occurs over TCP to ensure that the data arrives in order and in full (or not at all). This initial handshake is important, so reliable transfer over TCP is a must, even though most, if not all, subsequent communications will occur using UDP.

When a client device comes online, it connects to the known show control server's IP address and port. The server accepts and establishes a TCP connection. Once the connection is ready, the client sends a `DeviceID` identifying the specific client as

well as a `DeviceType` which identifies the type of hardware the device represents, such as an Operabot or Wall. The server checks to make sure that device's `DeviceID` has not previously connected. If it has not, the server dedicates a new multicast port to that `DeviceID`. If it has, the server reuses the previously assigned port. The server subscribes to the multicast group, and sends a final response over TCP telling the client to join the multicast group on the given port. The multicasts all use the same IP address (currently 224.0.0.1). The classes `ServerDeviceProtocol` and `ClientDeviceProtocol` implement this protocol for the server and client, respectively.

3.4.3 Server

Just like playback, the server must meet high performance expectations. It needs to handle at least twelve active devices—nine Operabots, three Walls, and maybe more. There are two connections per device as long as the TCP connection remains open. Moreover, there may be other clients besides the devices connected to the server. The Simulator, for instance, operates as a separate process rather than inside the show design and control software. It uses a network connection to receive updates for the virtual Operabots. Other components in the show may be interested in receiving updates as well. The audio system might want to know the position of specific robots so it could position sound sources as if they were emanating from those positions on stage.

Java supports two different methods of network I/O: sockets and NIO. Java's sockets allow for simple networking. Most methods on sockets block. Therefore, they are typically managed by a read thread and a write thread. This solution scales poorly. Twelve connected devices with two connections per device would result in forty eight threads! As more clients connect, that number quickly grows unmanageable. Java therefore offers NIO (New I/O) with support for non-blocking

I/O and native, low-level networking support. In Java, this is the only built-in method to efficiently manage large numbers of connections.

The server is built using NIO because of the performance requirements. This choice comes at a cost: the NIO API is significantly more complex. In brief, NIO uses a dedicated thread to respond to incoming and outgoing data. This single thread scales elegantly as the number of connections increases. The thread works in a tight loop, polling for updates.

NIO operates around communication pipes called `Channel`s. In particular, the high performance I/O uses a specific type of `Channel` called a `SelectableChannel`. Both `DatagramChannel` (UDP) and `SocketChannel` (TCP) implement `SelectableChannel`. `Selector`s maintain these `SelectableChannel`, which are uniquely identified by a `SelectionKey` for that `Selector`. Whenever the state changes on a `SelectableChannel`, such as when there are data to be read, the `Selector` adds that `Channel`'s `SelectionKey` to its list of keys to be processed. When a `SelectableChannel` becomes readable, for instance, its `SelectionKey` is added to the `Selector`. When the dedicated thread checks to see if any keys need processing, it finds a `SelectionKey` marked for read. Then, the thread performs a non-blocking read on the `SelectableChannel`. If the server needs to message a client, it must manually add a write request to the `Selector` by marking a specific `SelectionKey`.

The base NIO functionality is implemented in the class `AbstractNetworker`. Most of this functionality is in common between servers and clients. Therefore, the general `Client` and `Server` classes both extend `AbstractNetworker`.

Thread safety is of the utmost importance for the `AbstractNetworker`. Only the dedicated networking thread should ever touch the `Channel`s or the `Selector`. As well, performance is important. Therefore, much care has been given to ensuring

thread safety yet avoiding potentially slow concurrency control like locking. Instead, any collections that are accessed by multiple threads use an appropriate thread-safe container such as `ConcurrentMap` implementations or `ConcurrentLinkedQueue`s. These containers support compound atomic operations. `ConcurrentMap`'s `putIfAbsent` is one such method. Thoughtful application of this procedure—essentially a test-and-set for a map—circumvented the need for locking entirely.

Threads also come into play when dealing with read data. The server thread copies the incoming data and passes it to a `Handler` associated with the receiving `Channel`. `Handler`s are an intermediary between the `AbstractNetworker` and the `Protocol` used to interpret the incoming data. `Handler` uses its global thread pool to run the incoming data through the `Protocol`.

The `Protocol` interface adds support for converting back and forth between protocol-independent `Message` or `MessageBundle` classes and byte data. `Message` is a data unit comprised of a method name and any number of arguments. `Message` is parameterized on the type of the method name for added generality and flexibility. `MessageBundle` is simply a list of `Message`s.

The server currently defines three different `Protocol`s: `ServerDeviceProtocol`, `ClientDeviceProtocol`, and `OSCProtocol`. The first two protocols were discussed previously in Section 3.4.2. `OSCProtocol` implements the Open Sound Control (OSC) protocol using the server's Protocol interface. OSC is currently used as the protocol for communicating with devices over UDP multicast because it handles the structural hierarchy within devices nicely. Along with the physical machine address, OSC includes a semantic procedure address such as "/operabot/drive/x" and associated arguments, such as "1.0". The implementation is based on a portion of Illposed Software's open-source Java OSC library

[12].

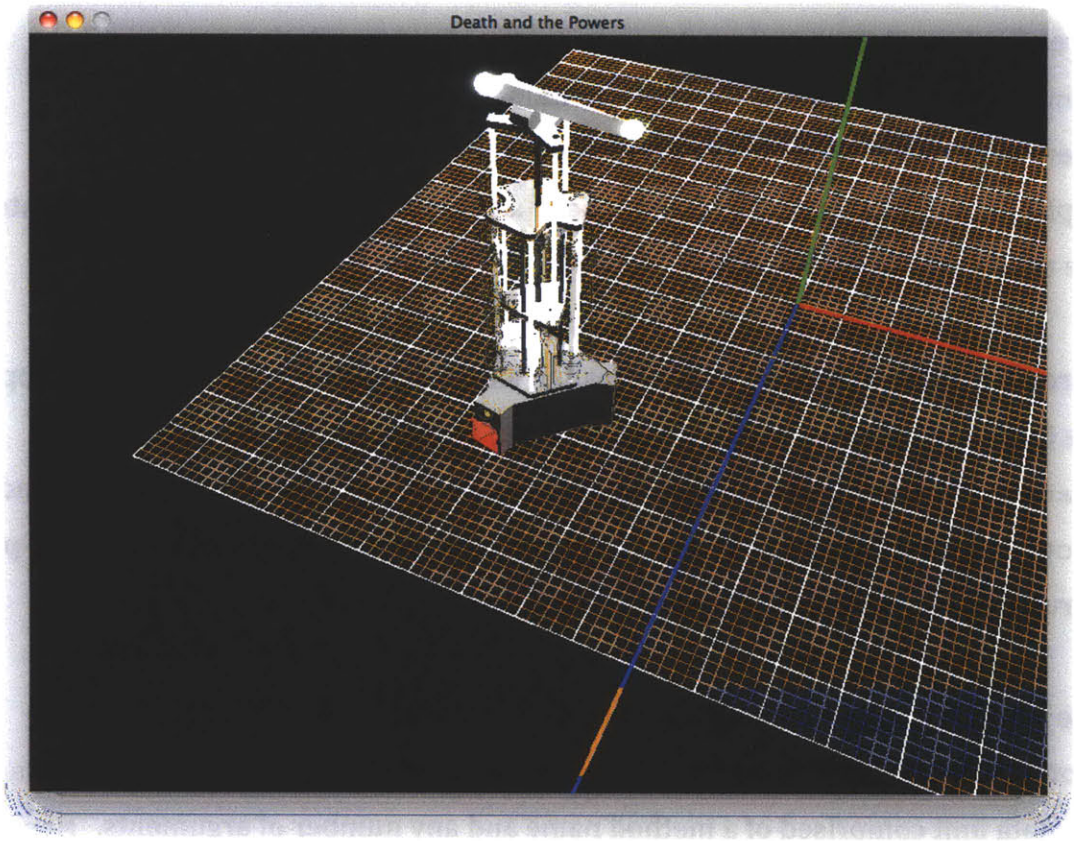

Figure 10. The Simulator showing an Operabot with lighting

3.5 Simulator

The Simulator visualizes the connected devices and their behavior based on the values sent by the show control system (Figure 10). The Simulator has two purposes. During the design process, it interacts with the design environment to accurately realize the Sequence Editor's instructions on playback and hence provide helpful visual feedback. During the show, it also acts as a controller such that operators can see which operators are controlling which robots or interactively select a robot to control.

During the show, four operators actively monitor and control individual Operabots and Walls. Instead of four people huddling around a single display running off of the show control's computer, each puppeteer ideally is situated with a machine running the Simulator, hopefully within view of the stage. As such, the Simulator is a separate application, designed to run on multiple computers simultaneously.

The show control server creates a multicast group whenever a device such as an Operabot or Wall connects to the server. This multicast group is intended for communications relating to that device. The Simulator leverages this design decision. Each Simulator registers with the show control server at startup. Whenever a device joins, the server notifies the Simulator that a new device has connected and offers it the multicast group. This way, the Simulator simply listens for the incoming messages and uses the messages to update the visual simulation.

Taking control of a robot is a matter of informing the server of the change in the routine. The server then ceases to send values for the controlled axes, and the Simulator becomes responsible for sending messages itself instead. Since all these communications occur on the multicast group, all the members of that group are privy to any messages sent, including the show control server, the Operabot's computer, and four Simulators at the very least. These messages include feedback from the Operabots that can also be visualized, such as remaining battery power.

The Simulator displays in 3D using the Java Monkey Engine (jME) library. A perspective view of the scene is not always useful, so other "stationary" views, such as from the top, will be available in the final version.

In its current form, the Simulator only displays a single Operabot. The final version will load the relevant 3D models on demand and associate them with the appropriate physical devices. The robots themselves use a combination of primitives shapes and models created by Peter Torpey and exported from

Autodesk's 3DS Max[6]. The Operabot, for instance, is made up of objects for the base, the midsection, and the head. The light tubes are represented using primitive cylinders. Lights are displayed using both color intensity and a "bloom" visual effect. Although the lights do not emit actual light in the 3D scene, the white color and glowing effect are a reasonable representation.

4. Discussion

The results so far summarize a work-in-progress on its way to its final form. These results, referring simply to the application itself and the source code, entail a discussion of their efficacy in accomplishing the prescribed goals.

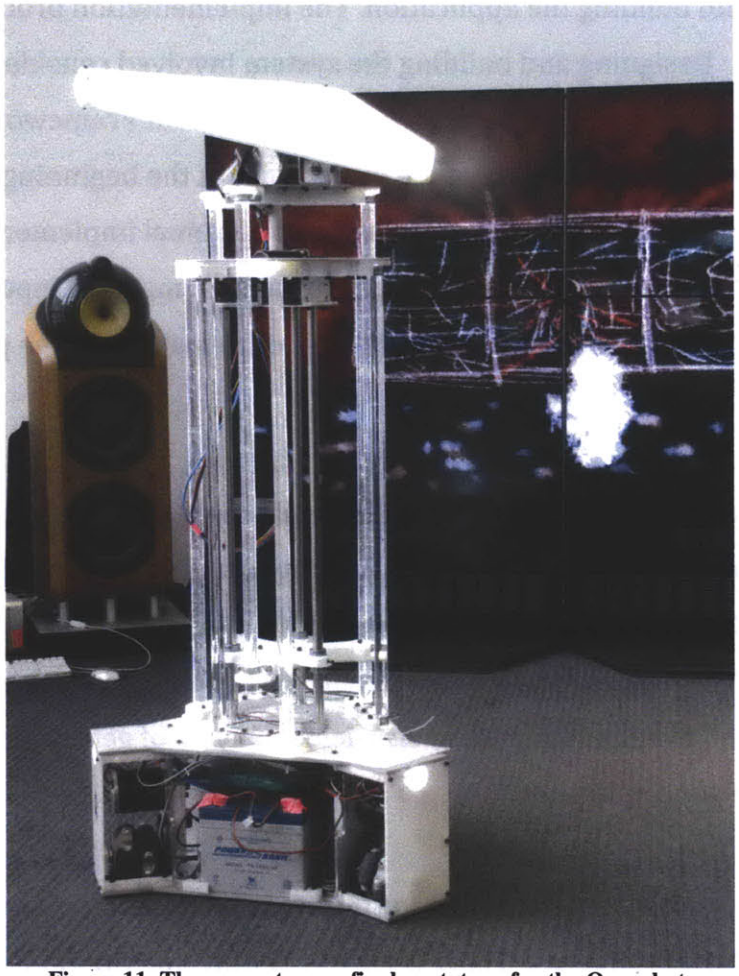

Figure 11. The current, near-final prototype for the Operabot

The details of the implementation itself are positive results of the process. The software works, allowing users to design routines for Operabots. It controls them as well, running the routines safely and accurately. The application accomplishes many of the goals and requirements outlined in Section 1 as well. The software as it

stands offers a stable and intuitive editing environment alongside a reliable and scalable server configuration.

As of writing, the show design and control application (along with the Simulator) consists of more than 20,000 lines of Java code, spanning over 260 individual files with at least as many Java classes. While this metric says nothing about the quality of the code, of course, it speaks to the shear amount of development time and effort that has gone into building the application. The implementation process has been highly iterative. Designing and building the system involved considerable backtracking and reworking of basic ideas. The Document Framework, for instance, evolved somewhere in the middle of the project, not at the beginning. The Property Pattern, similarly, arrived in response to extensive manual implementation of the Observer Pattern. These changes stemmed from group member input, evolving requirements and specifications, and maintenance to make sure the project could grow and adapt to new challenges easily.

The evolving nature of the codebase has led to an emergent structure that was not initially planned thanks to the Document Framework. The Document Framework and the Property Pattern are the generic scaffolding for any user-facing application, large or small. This application framework provides nearly automatic file I/O, undo-redo support, and event notifications. As such, it may be of general interest and use outside this specific application.

There are many unresolved issues left for future development before rehearsals begin and the show premieres. There is currently only one Operabot and no Walls, so testing has been relegated to a single device. Extensive testing has been done with this lone Operabot, however. The software has been employed to create routines and control the robot for many demos. These demos include joystick-controlled "dances" to music from *Death and the Powers* as well as more interactive scenarios where the operator guides the robot to interact with people as if it were

self-aware. In these real-world use cases, the software has performed admirably. The Operabot connects seamlessly to the server and responds to commands with no noticeable lag for either the live inputs or the predefined value curves.

The code has also proven itself to be very robust. The software has yet to fail during these demos, even when run for days at a time. While this testing does not offer nearly adequate coverage, these preliminary results are quite promising.

As a point of reference, consider the original control software this system replaces. In 2008, there was a rudimentary application to control the Operabots. It was capable of controlling the Operabot and its various axes. This application was buggy, unreliable, and not scalable. The user interface simply displayed information about the axes in a table and provided buttons to switch to different cues. This old software was a cause of consternation because it rarely cooperated with the software on the Operabot itself. The control software lagged and without any apparent cause. The robot would respond to commands anywhere between instantaneously and ten seconds later. This unreliable behavior made operating the robot in public both nerve-wracking and a liability.

In contrast, the show design and control software has instilled in the opera team a new sense of confidence about the robots and their capabilities. The software is robust and responsive where its predecessor was unstable and slow. The vastly improved performance owes much to the control software, along with improvements in the software onboard the Operabot, and the mechanical design.

While the software accomplishes many of its goals, such as providing a usable set of design tools, still much remains to be done. These shortcomings will be addressed in the near future as the project continues towards the premiere of the opera.

5. Next Steps

The opera opens in Monaco at the end of September 2010. While the groundwork here paves the way for the bulk of the application, much still needs to be done. Many features will be added to fill out both the design and show control aspects of the application.

5.1 Expanding the User Interface

Most design work for the user interface has been spent building the Sequence Editor. This emphasis is appropriate since the Sequence Editor is the primary view for interacting with the choreographic content of the show. However, there are a number of supporting interface elements that are necessary.

The show control application only supports one sequence currently. While one could theoretically work with multiple sequences and open multiple windows, there is no mechanism to do so in the UI. Similarly, while the Sequence Editor supports and even promotes reuse of curves, there is no way to simply view all the available curves. The UI should provide tools to manage all the assets used in the show so that a designer can access all the resources he or she needs.

Manipulating tracks in a sequence using the UI is not possible either. While a lot of attention and support has gone to interacting with clips and curves, manipulating the order and nesting of tracks is not currently supported. Tracks need some notion of selection just as is already available for clips and curves. Ideally, dragging a track's info pane should reorder them just as if they were elements in a standard UI tree control. As well, the interface should let users manipulate the properties of tracks. Users can rename tracks, but they should also be able to assign tracks different colors for organizational purposes or remap an individual track to a specific axis.

The user interface also lacks a high-level view for controlling and managing the progress of the show. This view includes an emergency stop, information about the current playback position and sequence, and feedback from the various connected devices. The devices should be sending back updates about their status or any errors they encounter so that operators can monitor the aggregated state of the system.

Lastly, the XML show file format currently only accommodates the `Document`s and `DocumentElement`s. A comprehensive format should also include information about the view state when the user saved the show. Windows open before should return to the same state and view as when the file was saved. While not strictly necessary for operation, this adds a level of professional polish.

5.2 Triggers in Sequences

Aligning time-dependent playback to live music is a difficult problem. One possible solution entails score following, a process that attempts to stay synchronized between a computer representation of the score (in MIDI, for instance) and the live playing. This method is technically complicated and rigidly bound to the underlying piece of music. Another method might simply attempt to find the tempo of the music by looking for prominent attacks in the audio. The tempo defines the relation between the musical representation in terms of beats and the real-world representation in terms of seconds. This sort of tempo analysis works very well with regular, percussive music like rock or pop. Unfortunately, it is ill-suited for contemporary art music.

In contrast to automatic methods like score following and tempo analysis, triggering is a manual method for timing alignment. Triggers offer a hybrid mode between live

control and fully defined value curves. Tracks listen for specific input actions to occur. These actions, then, trigger not simply a value change but an entire subroutine. For instance, it might be useful to attach an entire sequence to button presses on a joystick or key presses on a MIDI keyboard. In particular, the keyboard approach would allow musicians to accurately trigger timing-crucial moments along with the music.

These triggers need not apply only to little subroutines. Triggering is also a way of providing a degree of live control over the timing of the playback. Triggers could cause entire sequences to begin at a certain time, for instance. At a finer granularity, all the Operabots could begin a movement sequence when cued by the keyboard.

5.3 Integrating the Simulator

The Simulator is currently incomplete. While the basic premise works, it only supports one Operabot. Moreover, the server does not currently support clients like the Simulator that need to be notified when new devices connect. As such, the Simulator is simply a thin client that receives OSC messages over UDP. The final version will be a full-fledged client of the Server with the multicast methodology described previously.

The controller aspects of the Simulator are similarly unimplemented. During the early stages of implementation, a joystick could control the single Operabot. However, that feature disappeared when the Simulator became a network client. This support needs to be re-implemented. As well, the Simulator will need to talk to the show control server to make sure the two do not both take control of the same robot.

More work is also needed to make the Simulator a useful design tool. It currently only offers a perspective view. Other fixed viewpoints are used in design tools, such

as a top-down view or front view. These sorts of graphical changes are small and can be easily added but they are absolutely necessary features for usability. A more difficult problem, however, is adding the ability to use the Simulator to draw paths for the robots. It is hard to display a robot's position in a meaningful way using the Sequence Editor. A curve for X position and Y position, while straightforward and accurate, would be impossible for a user to picture mentally. Interacting with such a representation would be unintuitive and would stifle the creativity that the editing toolkit needs to encourage. Instead, the Simulator will have path drawing capabilities (much like the curve editing tools) that provide an intuitive design tool for defining movement.

6. Conclusion

Tod Machover's opera *Death and the Powers* presents many exciting technological innovations, including the Greek chorus of Operabots; "The Chandelier;" the three 14' tall robotic "Walls;" and more. The show design and control system described within and implemented through this project attempts to unify these many elements in ways that are both dramatically effective and safe. It is an application for both creating content and running the show itself.

The show design and control software was created for the explicit purpose of controlling the robotic elements of the opera. Previous proprietary tools could operate complicated stage machinery but none of them were well-suited for choreographing a troupe of robots. New software had to be created. While this software is designed and intended for robots in *Death and the Powers*, it represents a much more general ideal. The show design and control software extends beyond Operabots and the opera specifically. As new technologies arise, they will again need to be integrated into the theater environment. This show design and control software helps bridge the gap between technology and theater.

Acknowledgements

I would like to thank Tod Machover for his generous support and encouragement in this endeavor: you have given me an incredible opportunity working on the Operabots. This has been an incredible journey so far and it's only going to get better! Thank you, Peter Torpey, both for pushing me in new directions and for putting up with my constant (affectionate) teasing. And thank you, Elly Jessop, for patiently listening to my incessant whining and for always being a voice of sage wisdom. Ben Bloomberg: as always, you brighten my life and remind me that this bunch of trees is actually a forest. Bob Hsiung, Karen Hart, Donald Eng, and Gavin Lund—you are an incredible bunch of engineers and it's been an absolute pleasure working on the Operabots alongside you and making all sorts of nerdy jokes that shall go undocumented (for our own safety). To everyone else in the Opera of the Future group—Noah Feehan, Andy Cavatorta, Wei Dong—you guys are incredible and I cannot wait to see where you go. I would also like to thank my mother Sarah Minden for taking the time to proofread a document that probably sounded like it's in a foreign language. Lastly, I am hugely indebted to Shanying Cui, my loving and supportive girlfriend, who has encouraged me throughout this entire process and helped me up when I was down: you are my muse.

Bibliography

[1] Personal Robots Group. [Online]. http://robotic.media.mit.edu/projects/robots/operabots/overview/overview.html

[2] Ubisense. [Online]. http://www.ubisense.net/

[3] The Player Project. [Online]. http://playerstage.sourceforge.net/

[4] Microsoft Robotics. [Online]. http://www.microsoft.com/robotics/

[5] Stage Technologies. [Online]. http://stagetech.com/echameleon

[6] Autodesk. [Online]. http://www.autodesk.com

[7] Medialon. [Online]. http:// http://www.medialon.com/products/manager_5.htm

[8] Apple. [Online]. http://www.apple.com/finalcutstudio/finalcutpro/

[9] MOTU, Inc. [Online]. http://www.motu.com/products/software/dp/

[10] Entertainment Services & Technology Association (ESTA), Entertainment Technology - Architecture for Control Networks, 2006.

[11] Real-Time for Java Expert Group. Real-Time Specification for Java. [Online]. http://www.rtsj.org/specjavadoc/book_index.html

[12] Illposed Software. Java OSC. [Online]. http://www.illposed.com/software/javaosc.html